Fulfilling End-to-End Latency Constraints in Large-scale Streaming Environments

Stamatia Rizou, Frank Dürr, Kurt Rothermel
Universität Stuttgart, Institute of Parallel and Distributed Systems, Universitätstraße 38, 70569 Stuttgart, Germany
<firstname.lastname>@ipvs.uni-stuttgart.de

***Abstract*—The on-line processing of high volume data streams is a prerequisite for many modern applications relying on real-time data such as global sensor networks or multimedia streaming. In order to achieve efficient data processing and scalability w.r.t. the number of distributed data sources and applications, in-network processing of data streams in an overlay network of data processing operators has been proposed. For such stream processing overlay networks, the placement of operators onto physical hosts plays an important role for the resulting quality of service—in particular, the end-to-end latency—and network load. To this end, we present an enhanced placement algorithm that minimizes the network load put onto the system by a stream processing task under user-defined delay constraints in this paper. Our algorithm finds first the optimal solution in terms of network load and then degrades this solution to find a constrained optimum. In order to reduce the overhead of the placement algorithm, we included mechanisms to reduce the search space in terms of hosts that are considered during operator placement. Our evaluations show that this approach leads to an operator placement of high quality solution while inducing communication overhead proportional only to a small percentage of the total hosts.**

I. Introduction

Many modern applications relying on real time data such as network monitoring [6], multimedia streaming [10], and global sensor networks [3, 4], require the on-line processing of large streams of data from a distributed set of data sources. Imagine, for instance, a large-scale camera network that processes images from distributed data sources to detect activities inside buildings or across road segments. In this use case, large chunks of data are to be transmitted from the sources in order to get processed and finally delivered to the application. For such applications, the amount of data that is in transit in the network can be a hindrance for the scalability of the system since it could lead to traffic congestion and bottlenecks. Therefore, reducing the data load put onto the network is important for alleviating the system from unnecessary network load. Furthermore, many applications have stringent quality of service constraints, in particular with respect to end-to-end latency.

Distributed in-network processing of data streams in operator overlay networks is a promising method to solve these challenges. With this method, stream processing is modelled as a logical graph of operators that are placed on a set of hosts for execution. Data is streamed from the distributed data sources via chains of operators to the applications (sinks), and every operator performs a certain processing step on the data. Obviously, the placement of operators onto physical hosts affects the performance of the system with respect to scalability and the quality of service offered to the application. As already mentioned above, in particular the network load and end-to-end latency are important factors, which are in the focus of this paper. Formally, we consider operator placement as a constrained optimization problem, where the network load has to be minimized while fulfilling a given latency constraint defined by the application.

Operator placement is not a completely novel problem, and several placement algorithms have already been described in the literature. On the one hand, algorithms such as [11] and our previous work in [13] try to solve an unconstrained optimization problem without considering QoS constraints. Obviously, these approaches are only suitable for certain applications without stringent latency constraints. On the other hand, placement algorithms considering latency constraints and an optimization criteria have been proposed. Some approaches such as [8] consider different optimization criteria, in particular fair load balancing. Although load balancing also strives to improve scalability, it does not explicitly minimize network load. Instead we directly minimize network load using the available resources without striving for strict fair load balancing (only avoiding overloaded resources). In our previous work [12], we have already introduced an algorithm fulfilling latency constraints while optimizing for network usage. However, the previous algorithm only considered network delays in the calculation of the end-to-end latency, making it only suitable for applications where the processing delay can be neglected.

Our approach first optimizes for network usage and then applies a constraint satisfaction algorithm that fulfils the end-to-end latency constraints. The computing resources are used in an efficient way in the sense that nodes with more residual resources are preferred against others, only if they reduce the processing delay of the corresponding operators to be placed. Our evaluations show that by the use of simple heuristics that restrict the search space, we can achieve good quality solutions by considering only 5% of the total hosts.

The rest of the paper is structured as follows. In Section 2 we discuss the related work. In Section 3, we introduce our system model and define formally our problem. In Section 4 we describe the general course of actions of our placement algorithm. Finally, in Section 5 we present our experimental results, before we conclude in Section 6.

Published in International Performance Computing and Communications Conference @IEEE 2011

II. Related Work

The operator placement problem has been investigated in different contexts. In [4] an approach to serve multiple queries with different resolutions has been proposed. This work focuses on the reuse of operators and the different levels of data granularity. An optimized strategy for sharing operators is out of the scope of this paper.

Closer to our specific goals, the minimization of the network load without considering QoS constraints has been one of the main goals of other existing placement algorithms. Network load is frequently expressed as the network usage of inter-operator data streams based on the bandwidth requirements of inter-operator data streams and the delay between operators. Pietzuch et al. presented the first approach, called SBON, to optimize network usage using a spring relaxation model [11]. In [13], we presented a different algorithm to minimize the bandwidth-delay product of inter-operator data streams that further reduced the network usage. This approach first tries to find an optimal solution in a continuous search space (latency space; see below) and then maps this solution to the discrete set of hosts. We apply this algorithm in the enhanced algorithm of this paper to calculate an unconstrained optimum that is then modified to satisfy the latency constraint.

Since latency is a crucial QoS parameter, many placement algorithms try to achieve latency guarantees. From the complex event processing field, Cordies [9] has been proposed for efficient distributed event correlation. Cordies focuses on the fulfilment of latency constraints and does not consider network load as an optimization goal. Other approaches combine latency guarantees with load balancing. Gu et al. presented an algorithm that uses global knowledge to checks exhaustively all hosts in order to identify some candidate hosts [8]. Then, it selects the hosts that minimize a congestion aggregation metric modelling the processing and network residual resources. Such an optimization metric can be useful for cases where the network is heavily loaded, but it is less efficient for other situations.

In contrast to these approaches that consider latency constraints together with load balancing, we do not strive for fair load distribution, but only try to avoid overloaded hosts while defining minimal network usage as our primary optimization goal. In our previous work [12], we have already considered the problem of optimizing network usage while providing latency guarantees. However, in this previous work we only targeted a subset of applications, for which the processing delay does not contribute significantly to the total end-to-end latency. Here, we overcome this restriction by new placement algorithms explicitly taking the processing delay into consideration.

III. System Model and Problem Statement

Next, we introduce our system model. Then, we give a formal definition of our operator placement problem.

A. Execution and Data Model

In our execution model, a stream processing task is modelled as a directed *operator graph* $G = \{\Omega, S, A, E\}$. Ω denotes the set of *operators*. Operators include *data sources*, *applications* (*sinks*), and *processing operators*. $S \subseteq \Omega$ and $A \subseteq \Omega$ denote the sets of data sources and applications (sinks), respectively. $E \in \Omega \times \Omega$ is a set of links (directed edges) that connect operators.

In order to define the end-to-end latencies (see below), we need to introduce the notion of an *end-to-end path* $\overline{\omega_i \omega_j}$. Each end-to-end path connects a source operator $\omega_i \in S$ with one of the sinks $\omega_j \in A$. $\Phi(G) = \{\overline{\omega_1 \omega_i}, \ldots, \overline{\omega_j \omega_n}\}$ denotes the set of all end-to-end paths of the operator graph G. Note that an end-to-end path contains all the operators on that path as well as the links between these operators. We write $\omega \in \overline{\omega_i \omega_j}$ if operator ω is part of the path $\overline{\omega_i \omega_j}$; we write $\omega_k \omega_l \in \overline{\omega_i \omega_j}$ if the link $\omega_k \omega_l$ is part of path $\overline{\omega_i \omega_j}$.

In our data model, data is organized in minimal discrete data units called *tuples*, denoted as τ. A sequence of tuples forms a data stream. Tuples can be as small as a single number, e.g. a temperature value, or as big as a chunk of several megabyte of data, e.g. an image from a camera. We define s_τ as the size of tuple τ. A tuple forms the basic unit of processing for each operator. Sources generate typically sequences of tuples in intervals that are then processed by operators and finally consumed by the sinks.

Operators are placed on physical hosts, denoted as ν. We assume that sources and applications are pinned to certain hosts, whereas every other operator can be placed freely on any hosts. After operator placement, the operator graph can be interpreted as an *operator overlay network*. Next, we define the end-to-end latency and network load resulting from a certain placement.

B. End-to-End Latency

We define the end-to-end latency $L(\tau, \overline{\omega_i \omega_j})$ of path $\overline{\omega_i \omega_j}$ as the time that tuple τ needs to get transmitted and processed along a path between source ω_i and sink ω_j:

$$L(\tau, \overline{\omega_i \omega_j}) =$$

$$\sum_{\omega_k \omega_l \in \overline{\omega_i \omega_j}} \{N(\overline{\nu_k \nu_l}) + T(\tau, \nu)\} + \sum_{\omega_k \in \overline{\omega_i \omega_j}} P(\omega_k, \nu)$$

In this equation, $N(\overline{\nu_k \nu_l})$ defines the *propagation delay* of a link $(\overline{\nu_k \nu_l})$ on the path, i.e. the time it takes to transmit a single bit via UDP/IP or TCP/IP between the two physical hosts hosting operator ω_k and ω_l. For modelling the propagation delay, we use a model of the physical network called *latency space* introduced by Pietzuch et al. in [11]. According to this model, each physical host ν is assigned a set of coordinates in Euclidean space, such that the distance $d(\overline{\nu_i \nu_j})$ between two physical hosts ν_i, ν_j models the propagation delay $N(\overline{\nu_i \nu_j})$. The model is created and constantly updated using round trip time measurements between hosts. In that sense, the distance of two operators in the latency space models the expected propagation delay after the mapping of operators to physical hosts. We assume this latency model to be available to every host running our distributed placement algorithm, for instance, using a centralized service or distributed peer-to-peer system managed by the physical hosts. This systems has to support queries for the coordinates of individual hosts as well as range queries for all hosts in a certain part of the latency space.

$T(\tau, \nu)$ defines the *transmission delay* for putting a tuple on the wire at host ν hosting ω_k. In order to estimate $T(\tau, \nu)$, we measure the transmission delay $T(\tau', \nu)$ of a real tuple τ' of size s'_τ on a physical host ν and we calculate the transmission delay as $T(\tau, \nu) = (s_\tau / s'_\tau) T(\tau', \nu)$.

$P(\omega_k, \nu)$ defines the *processing delay* that a tuple experiences at host ν hosting operator ω_k. As presented next, we use a simple processing model that assumes that the processing power of a host is equally distributed to all operators running on this host. We have evaluated this processing model on different machine types, and the approach was able to keep the relative error lower than 17% on average—a value sufficient for our purpose. However, due to space limitations, we leave out the evaluation results of the processing model. Alternatively, other processing models can also be applied.

For the local host where the operator is currently located, $P(\omega_k, \nu)$ could be measured directly. However, determining $P(\omega_k, \nu)$ is not a trivial task for other hosts where the operator is currently not located at. Note that the placement algorithm needs information about $P(\omega_k, \nu)$ before it actually places the operator on host ν to make a decision which host is suitable w.r.t. processing delay before actually migrating an operator. Therefore, the basic problem is to *estimate* the processing delay of an operator ω when executed on host ν taking into consideration the fact that host have dynamic processing load and different processing power.

Our estimation is based on two metrics to define the processing power and load of each host, respectively. On the one hand, we use the *bogomips* metric to define the speed of a machine. Bogomips expresses the number of iterations per second of a loop with empty body. Obviously, this metric cannot capture every aspect of the speed of a host such as different relative speeds for integers and floating point operations. However, it gives a coarse estimate to compare two machines and proved to be sufficiently accurate in our measurements. On the other hand, we use the run queue length of the processor to express the load of an host. The run queue length defines the number of processes waiting for the CPU. Intuitively, the share of processing time an operator receives will shrink proportional to the number of processes running on the host (here, a process can be another operator as well as any other process running on the host).

Assume that the operator is currently running on host ν_i and we want to estimate the processing delay of that operator if it migrates to host ν_j. The current capacity c_{ν_i} of host ν_i with processing speed $mips_{\nu_i}$ and run queue length q_{ν_i} is given by the following formula: $c_{\nu_i} = \min\{mips_{\nu_i}, \frac{mips_{\nu_i}}{q_{\nu_i}}\}$. The capacity of the other host is given by: $c_{\nu_j} = \min\{mips_{\nu_j}, \frac{mips_{\nu_j}}{q_{\nu_j}+1}\}$. Here, $mips/q$ defines the bogomips that one process receives if q processes are competing for the CPU. On host ν_i where the operator is currently placed, q_{ν_i} already includes the operator. On the (potential) host ν_j we have to add 1 to q_{ν_j} to reflect the queue size after the migration to ν_j. The minimum function ensures that on an unloaded host and short processing times with longer idle periods between tuples the operator cannot receive more than 100% of the CPU. We also designed extensions of these formulas for multi-core CPUs, which are not further discussed here due to space restrictions. As an indicator of the current relative performance of the two hosts we define the *speedup factor*: $speedup_{ij} = \frac{c_{\nu_i}}{c_{\nu_j}}$. Finally, we approximate the remote time to run the operator on host ν_j as the product of the speedup factor and the local processing time at hoste ν_i: $P(\omega, \nu_i) = speedup_{ij} * P(\omega, \nu_j)$.

Based on the previous definitions, we can define the latency of an operator graph as the maximum end-to-end latency contained in operator graph G, i.e. the maximum latency that a tuple experiences traversing the longest path in the operator graph. Formally speaking, the latency of an operator graph G is defined by:$L(G) = \max_{\overline{\omega_i\omega_j} \in \Phi} L(\tau, \overline{\omega_i\omega_j})$

C. Network Usage

Apart from the end-to-end latency specifying the desired QoS, we need to formally define the network usage that represents our optimization metric. The network usage $U(\overline{\omega_i\omega_j})$ of a single link is defined by the bandwidth-delay product of that link: $U(\omega_i\omega_j) = r(\omega_i\omega_j)N(\omega_i\omega_j)$. Here, $r(\omega_i\omega_j)$ denotes the data rate between two operators ω_i and ω_j and $N(\omega_i\omega_j)$ the propagation delay of that link. Intuitively, this product indicates the load of an overlay link in bits that are in transit in the network on that link. The network usage of a complete operator graph G is defined as the sum of the network usage of all links of the graph: $U(G) = \sum_{\omega_i\omega_j \in \Phi} r(\overline{\omega_i\omega_j})N(\omega_i\omega_j)$.

D. Problem Statement

Based on the end-to-end latency and network usage definitions, we can now formally define our placement problem. This problem is defined as constrained optimization problem where a user defined maximum end-to-end latency restriction R has to be fulfilled while minimizing the induced network usage:

$$U(G) = \sum_{\omega_i\omega_j \in \Lambda} r(\omega_i\omega_j)N(\omega_i\omega_j) = min \qquad (1)$$

$$\text{s.t. } L(G) \leq R \qquad (2)$$

IV. Placement Algorithm

In this section, we present the operator placement algorithm to solve the above constrained optimization problem. We start with an overview of the algorithm and then present further details in the following subsection.

A. *Overview of Algorithm*

The basic idea of the algorithm is to use a two-step placement process. In the optimization step, we search for an optimal placement w.r.t. network usage. In the second step, we modify this unconstrained solution such that the latency constraint is satisfied and the network usage is only increased as few as possible compared to the unconstrained solution. Both steps are executed in two sequential phases. Firstly, in the *initial operator placement phase* to obtain an initial mapping of operators to hosts. Then both steps might be repeated during the *adaptation phase* to adapt to changing network conditions either violating the QoS constraint or rendering the initial solution suboptimal.

Although the optimization step and constraint satisfaction step are both executed in both phases, the hosts responsible for the execution of these steps are different for the initial placement phase and adaptation phase. For the initial placement, a *coordinator* node is responsible to find an initial placement for each operator and deploying the operators on hosts. The coordinator node can be any node, e.g., the node that initially received the query. During the runtime of the subsequent adaptation phase, the operators are monitored, and particular network changes trigger the re-placement of operators. In this case, the solution of the unconstrained optimization problem is found in a distributed manner by the nodes hosting operators of the graph, while after the optimization step the solution of the constrained optimization problem is calculated again on a coordinator node. For the adaptation phase, the coordinator node is the node that hosts the root of the operator graph such that communication overhead is reduced.

For solving the unconstrained optimization problem, we use the algorithm presented in [13]. This algorithm can either be executed centrally on the coordinator node (Phase 1) or in a distributed manner (Phase 2) based on a local view of each operator onto data rates of incoming and outgoing streams and propagation delays to graph neighbours. After a number of iterations, the algorithms converges to an optimal solution of the whole operator graph. The output of the algorithm, is the optimal position of the operators in the latency space. For a more detailed description of the unconstrained optimization algorithm, we refer to [13].

Starting from this optimal placement, we distort the optimal solution minimally to satisfy the latency requirements of the user. Intuitively, this means that the operators should be placed on hosts that reduce the end-to-end latency, either by moving to faster nodes (reducing processing delay) or by reducing the network latency. Theoretically, we could find the optimal solution of the constraint placement problem by an exhaustive search that considers every host in the system. However, obviously this would lead to high overhead for larger sets of hosts and operators. Therefore our solution is based on the idea to find some *candidate* hosts that reduce the end-to-end latency. We find promising nodes by a search in certain areas of the latency space—later we will show in detail how to find a good set of candidates. Then, we communicate with the candidates to get their processing and transmission delay. Finally as we see later, in order to keep the network usage as low as possible, we iterate over the candidate nodes and we select those that reduce the end-to-end latency while increasing the network usage minimally.

Depending on the phase, the output of the constraint satisfaction algorithm will be either an initial placement or a new placement of the operators. In the latter case, the operators are migrated to the new hosts.

B. *Constraint Satisfaction Algorithm*

Next, we describe the details of the constraint satisfaction step. As mentioned, the constraint satisfaction algorithm depicted in Algorithm 2 is invoked after the optimization step. Therefore, before the execution of this algorithm all operators are placed on hosts such that Equation 1 is minimal. For the explanations below, it is important to realize that $U(G)$ is a function that depends on the coordinates of the hosts hosting operators in the latency space since the Euclidean distance between hosts in the latency space defines the propagation delay (Function N) between hosts and therefore their operators. In the beginning, $U(G) = U_{\min}$ where $U_{\min}$ denotes the minimal network usage, which is found by the optimization step. However, although $U(G)$ is minimal after the optimization step, the latency of the longest path of the graph might be higher than the requested maximum latency, i.e., Equation 2 is not fulfilled in general. Algorithm 2 now tries to distort this optimal solution to stay as close as possible to $U_{\min}$ and fulfill the latency constraint.

Algorithm 2 gets as input an initial mapping of the operators to hosts such that the network usage of the operator graph is minimal [13]. Firstly, the algorithm finds the longest path in the operator graph, and checks if the latency restriction is already fulfilled. In that case, it simply returns the current mapping. Otherwise, it enters the main body of the algorithm, where it checks for alternative mappings. For each operator on the longest path, the algorithm finds a set of candidate hosts where the operator could be migrated to. The candidate set is calculated once in the beginning for each operator on a path. The candidates are selected such that moving an operator to a candidate host decreases the latency of the longest path. In the next subsection, we are going to discuss in detail how this candidate set is determined. If the candidate set is empty, the latency cannot be decreased any further and the algorithm stops without finding a valid solution. In this case, the application is notified that the

Algorithm 1 Constraint Satisfaction Algorithm

```
Require: $U(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$ is minimal
Ensure: Finds a mapping $(\nu_1, \ldots, \nu_n)$ such that $L(G) \leq R$
    and $U(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$ is minimal
 1: while $(L(G) > R)$ do
 2:   find maximum latency path $\overline{\omega_i \omega_j}$
 3:   if candidate set $candidates(\omega)$ does not exist then
 4:     for all operator $\omega \in \overline{\omega_i \omega_j}$ do
 5:       find candidate set $candidates(\omega)$
 6:       sort $candidates(\omega)$ by distance to $U_{\min}$
 7:     end for
 8:   end if
 9:   if $candidates = \emptyset$ then {latency minimum}
10:     notify application
11:   else
12:     for all operator $\omega \in \overline{\omega_i \omega_j}$ do
13:       get next candidate $\nu'$ in $candidates(\omega)$
14:       calculate the difference in network usage $\Delta U(\omega)$
15:     end for
16:   end if
17:   assign operator $\omega$ with minimal $\Delta U(\omega)$ to $\nu'$
18:   delete candidate $\nu'$ from $candidates(\omega)$
19: end while
20: return current mapping $(\nu_1, \ldots, \nu_n)$
```

latency constraint cannot be fulfilled, and the application might choose to decrease its requirements or simply stop. If the candidate set is not empty, latency can be further decreased by migrating to any candidate host. The idea is, not to choose an arbitrary candidate but a candidate that increases the network usage the least in order to distort the optimal solution w.r.t. to network usage the least. To this end, the hosts of the candidate set are sorted according to the distance to $U_{\min}$, and the host with the minimal distance leading to the minimal network usage increase $\Delta U(\omega)$ is chosen as new host for operator ω.

C. Selection of Candidates

Calculating the candidate set is a crucial operation during the constraint satisfaction step. If the candidate set is too big, the overhead increases since every candidate has to be contacted and checked with respect to its processing and network delay. If the candidate set is small and misses some valid hosts that would decrease latency, no valid solution might be found although it exists in the network. In order to find a good trade-off between overhead and success rate, we considered different candidate selection strategies.

First, we restrict the search space by filtering out the physical hosts according to their location in the latency space. Next, we illustrate this idea through a simple example and we prove an optimal pruning criterion that reduces further the search space. Firstly, it is important to observe that all suitable candidates are restricted inside ellipsoidal shapes in the Euclidean latency space. Figure 1 visualizes a simple example of an operator ω with one source ω_i and one sink ω_j. The end-to-end delay L for this simple example is the sum of the propagation delays of the operator to its neighbours $N = N(\omega\omega_1) + N(\omega\omega_2)$, plus the processing delay $P = P(\omega, \nu)$ at the host ν of operator ω, and the transmission delays $T = T(\tau, \nu) + T(\tau', \nu)$ of the input tuple τ and output tuple τ'. Each value of the end-to-end delay L represents an ellipse in the latency space with foci points the positions of the two neighbours ω_i and ω_j. Note that according to the definition of the ellipse, all points on the ellipse have the same distance to the foci points, i.e., they lead to the same end-to-end delay. It is straightforward to see that only the nodes that reside inside the ellipse $L = N + P + T$, can lead to better solutions since for nodes outside the ellipse even if the processing delay is zero, the network delay would still exceed the current latency L. Thus, the candidate nodes are restricted inside the ellipse L. In order to find the candidate hosts within L, we perform a range query in the latency space using the latency space service and query range L.

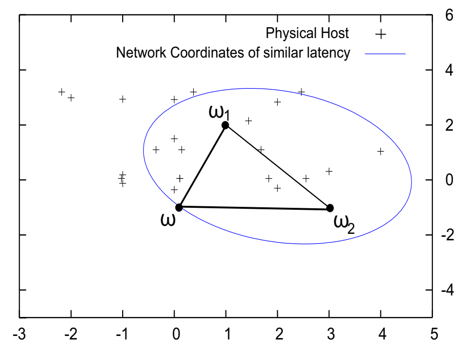

Figure 1. Candidate set for one unpinned operator.

Checking all nodes inside the ellipse cannot be a valid selection strategy, since the set of nodes might be significantly large and as a result would lead to high communication overhead. Moreover, since in a real system the load of the system changes frequently, asking a large set of nodes could degrade the information about the current load of nodes due to long delays. Thus, in order to limit the candidate nodes to a reasonable size, we select only k hosts in the ellipse to contact. Straightforward solutions to that problem are to choose the k closest nodes with respect to network usage minimum that reside in the ellipse, or to select k random hosts inside the ellipse. Random selection of hosts could be beneficial in case the suitable hosts do not lie in the direct vicinity of the current host. A more sophisticated method could be to use a pruning criterion that can filter out some of the nodes inside the ellipse. To this end, we introduce the following pruning criterion for the processing delay of the candidate hosts:

Algorithm 2 Candidate Selection Algorithm

```
Require: Bounding box for ellipse $E$
Ensure: Candidate set $candidates$ of size $k$
1: find all hosts $hosts$ inside ellipse $E$ [range query]
2: sort hosts $hosts$ by distance to $U_{\min}$
3: while $\#candidates < k$ do
4:     contact next host $\nu'$ in $hosts$
5:     if $P(\omega,\nu') < P(\omega,\nu) + T + N - N_{\min}$ then
6:         $candidates \leftarrow \nu'$
7:     end if
8: end while
9: return $candidates$
```

Pruning Criterion Let ω be an operator placed on a host ν with network latency N to two neighbouring operators and with processing delay P. Assume also a tuple τ with transmission delay T. An host ν' can only lead to a better solution than that of ν w.r.t. latency, if and only if the following condition is fulfilled: $P(\omega,\nu') < P(\omega,\nu) + T + N - N_{\min}$, where $N_{\min}$ represents the minimum network delay of operator ω to its two neighbours.

Proof: Assume that host ν' lies on the line segment between the two neighbouring operators ω_1 and ω_2. Note that a host on this line segment leads to the minimum possible propagation delay $N_{\min}$. Furthermore, in the best case, ν' will have negligible transmission delay, i.e., $T = 0$. In this case, the latency L' for a placement on ν' is equal to $P(\omega,\nu') + N_{\min}$. The latency L for the current placement on ν is $P(\omega,\nu) + N + T$. If it should hold $L' < L$, then $P(\omega,\nu') + N_{\min} < P(\omega,\nu) + N + T \rightarrow P(\omega,\nu') < P(\omega,\nu) + T + N - N_{\min}$ ■

Algorithm 2 shows the pseudocode for the candidate selection strategy using the pruning criterion. According to this method, we first get all the hosts that reside in the ellipse by performing a range query at latency space. Then we contact one by one the next nearest host with respect to network usage minimum inside the ellipse and we check if it satisfies the pruning criterion. In that case, the host is included to the candidate set. The process is repeated until k hosts that satisfy the pruning criterion are found. Obviously, this method induces higher overhead, than the naive solutions proposed earlier, but it is expected to give better quality results, since it takes also into consideration the processing delay. Thus, if the current host is quite fast, the criterion tends to filter out more hosts, while in case of a slow current host, less candidate hosts will be filtered out. In our evaluation, we show how these different strategies perform in terms of overhead and result quality.

V. Evaluation

In this section, we present our evaluation results. We start with a description of the evaluation setup. Then we evaluate in detail the performance of the placement algorithm in terms of optimality w.r.t. network load and its capability to satisfy latency constraints.

A. Setup

To evaluate our algorithms, we have implemented them for the NET cluster [7], an emulation environment developed at the University of Stuttgart. NET provides an emulation environment for testing distributed systems and communication protocols. It combines the benefits of real-time experiments and network simulation. NET consists of a compute cluster, where every cluster node hosts several virtual nodes (in our case the operator hosts) that execute real implementation of the “software under test”. Nodes are connected by an emulated communication network that can be parametrized such that it resembles a given network (including network topology and link characteristics such as latency and bandwidth). Using emulation instead of simulation gives us the chance to test a real implementation of our placement algorithm under realistic conditions.

Our emulated system consists of 200 hosts. Each host has a capacity of $4,800$ bogomips. To define the latencies and bandwidth between every pair of hosts, we used real measurements between PlanetLab[2]. To construct the latency space model containing the virtual network coordinates of hosts, we used the Pyxida system running on each host [1]. Pyxida implements the Vivaldi algorithm [5] in order to calculate accurate coordinates where the distance closely matches the propagation delay.

In order to vary the processing load induced by operators, we used operators implementing a matrix multiplication with different sizes. Besides giving us the opportunity to easily manipulate the processing load of operators, matrix multiplication also is a realistic operator is used, for instance, for traffic matrices in network monitoring, or image recognition. We varied the size of matrices in the range from 50 to 500 by defining four discrete sizes of $\{50, 100, 200, 500\}$ elements. Thus, we cover a large spectrum of heterogeneous operators in terms of processing load. Consequently, the size of the tuple is defined by the size of the matrices.

For our experiments, an operator graph, has typically two free operators to be placed. The data sources feed the operators with data every 20 up to 120 seconds following a uniform distribution, leading to heterogeneous data rates. Moreover, the data sources and sinks, are uniformly distributed on random hosts in the network. The parameter k that defines the size of the candidate set is set to 5 hosts, i.e. 2.5% of the total number of hosts in the network. We evaluated our placement algorithm with the four different candidate selection strategies presented in Section IV. *Ellipse* (EL) represents the strategy that checks all nodes inside the search ellipse. *K-Nearest Neighbour* (kNN) implements the k-nearest neighbour search for hosts inside the ellipse in the latency space. *K-Random* (kRand) implements the random selection strategy, which selects random hosts that

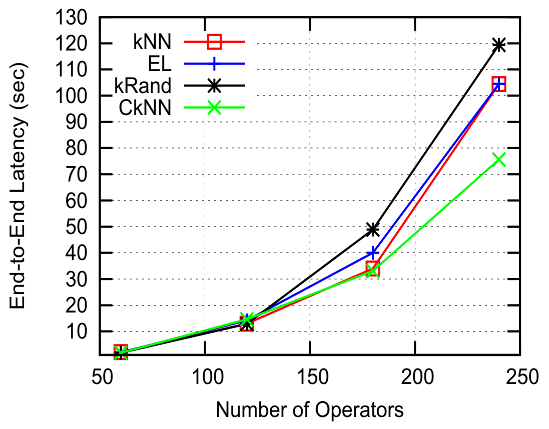

Figure 2. End-to-End Latency for increasing number of operators.

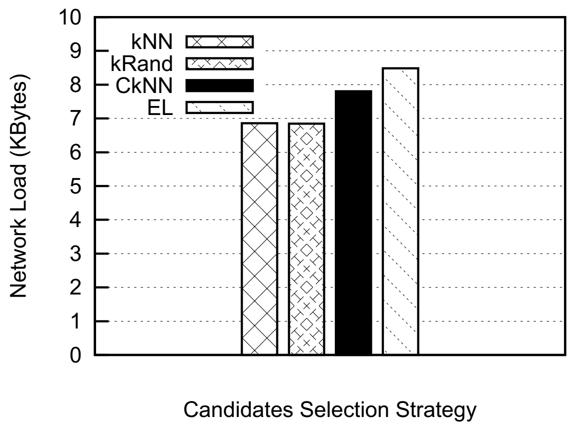

Figure 3. Resulting network usage for candidate selection.

reside inside the ellipse. Finally, *Conditional K-Nearest Neighbor* (CkNN) implements the pruned search according to the pruning criterion presented earlier. For each candidate selection strategy, we measure the resulting end-to-end latency, the network usage and the communication overhead to discover the candidate hosts.

B. *End-to-End Latency*

In the first experiment, we evaluate the QoS capabilities of our placement algorithm with the different candidate selection strategies. In order to explore the limitations of the different strategies, we consider an extreme case with very hard latency constraints: By setting the latency constraint to zero, we let the placement algorithm search for the operator placement with minimum possible latency. We deployed up to 240 operators gradually and measured the achieved latency for each candidate selection strategy.

Figure 2 shows the achieved latency over the number of deployed operators. As expected the latency increases with the number of deployed operators since the system load increases. Initially all methods almost perform similarly since initially the system has no load and all hosts can execute the operators with the same expected (low) delay. In this case, the solution is mainly defined by the network latency and not by the processing delay.

As the number of operators increases, some hosts get more load and become slower in comparison to other hosts. In that case, the latency of the random strategy kRand increases faster compared to the other strategies since it selects randomly hosts inside the ellipse. The greedy strategy kNN is more resilient to the load but finally deviates also significantly from the CkNN up to 38% since it only considers a limited set of hosts in the vicinity of the network usage minimum. Another interesting result is that the approach with the optimal restriction (EL) performs similar to the greedy kNN strategy, without achieving the best result. There are multiple reasons that could have degraded the performance of the expected optimal strategy. First, the processing and network model are designed to give a rough estimation of the delays and not very accurate estimations that the search of an optimal solution would need. Second, as we see later this method queries many hosts to decide on the next host and as a result it reacts slowly to network changes and it might even use outdated delay measurements.

C. *Network Usage*

Next, we analyse the optimality of the different candidate selection strategies in terms of network usage. As already mentioned before, the different strategies try to strike a balance between latency constraint fulfillment and optimality. Therefore, we expect the approaches with better QoS performance to have the higher costs in terms of network usage. For the same experiment as before, we calculate the average network load of the deployed operator graphs. In order to measure the network load, we have taken snapshots of the data that were in transit at certain points in time. Thus, we have calculated the amount of data that are in transit in $KBytes$. Finally, we have calculated the average data load over the time for the different strategies. Figure 3 shows the absolute values in $KBytes$ of the network load for the different candidate selection strategies.

As we see in Figure 3 the network usage is low for the greedy kNN strategy and the random strategy. That is expected, since these approaches do not fulfill optimally the latency constraints and therefore can achieve a lower network usage. Moreover, CkNN induces more network load, achieving although a good balance between the network usage minimization and the fulfillment of the latency constraints. Finally, EL does not manage to find good candidate hosts and it induces also high network load.

D. *Communication Overhead*

Finally, we discuss the communication overhead induced by each candidate selection strategy. Figure 4 shows the

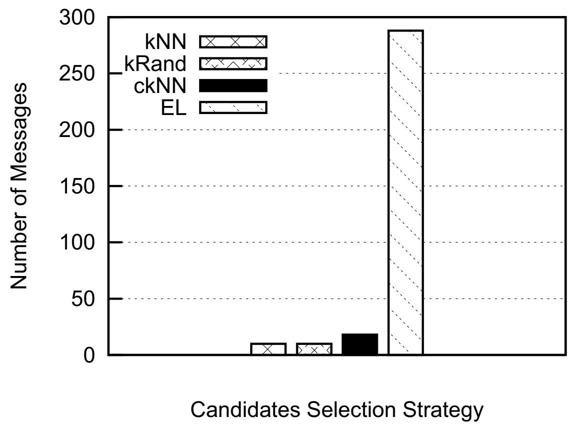

Figure 4. Communication Overhead for candidate selection.

average number of messages communicated between the coordinator and other hosts in order to define a candidate set. For kNN and kRand the number of messages are 10, since by default these strategies communicate with only $k = 5$ hosts and therefore they need for each contact two messages (request/response). For the EL algorithm using the optimal restriction in the latency space, the average number of messages is 288. This means that a host should contact on average 144 out of 200 hosts to decide on a placement. In practice, this method cannot be used not only because of the high communication overhead that induces, but also because it cannot guarantee high quality solutions, since it would react extremely slow at each network change using possibly outdated delay measurements.

Finally, for CkNN the number of messages is 18. Thus, we see that the strategy that uses the pruning criterion not only performs better in terms of constraint satisfaction problem, but also keeps the number of messages very low querying on average about 5% of the total hosts. In other words, we see that it is sufficient to check only a small subset of all hosts that reside in the ellipse.

VI. Summary and Future Work

In this paper, we focused on data stream processing systems that need to process high-volume data streams from distributed data sources under certain latency constraints. We motivated to use an overlay network of processing operators for the online processing of streams. As main contribution, we proposed a novel operator placement algorithm that distributes operators among a set of hosts such that the induced network usage is reduced and the given latency constraint is fulfilled. We first formulated the problem as a constrained optimization problem. We proposed a method for estimating the processing time of operators as a necessary prerequisite for finding suitable operator hosts in order to fulfill the latency constraint. Then, we proposed different selection strategies to restrict efficiently the search space. Our evaluations showed that the heuristic that uses the pruning criterion performs better in terms of quality of the solution over a greedy and a random selection strategy, while it invokes a small communication overhead.

As future work, it would be interesting to investigate the trade-off between migrating operators and achieving good quality solutions in highly dynamic environments. Thus, we could increase the resilience of our solution to frequent changes of the environment.

Acknowledgment

The work presented in this paper was supported by the German Research Foundation (DFG) within the Collaborative Research Center (SFB) 627.

References

[1] Network Coordinate Research at Harvard. http://www.eecs.harvard.edu/~syrah/nc/.

[2] Planetlab. http://www.planet-lab.org.

[3] K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *The 2007 International Conference on Mobile Data Management (MDM 2007)*, pages 198–205, 2007.

[4] A. Benzing, B. Koldehofe, and K. Rothermel. Efficient Support for Multi-resolution Queries in Global Sensor Networks. In *COMSWARE*, 2011.

[5] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM '04*, 2004.

[6] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better NetFlow. In *ACM SIGCOMM*, 2004.

[7] A. Grau, K. Herrmann, and K. Rothermel. Efficient and Scalable Network Emulation Using Adaptive Virtual Time. In *18th Internatonal Conference on Computer Communications and Networks*, Aug. 2009.

[8] X. Gu, P. S. Yu, and K. Nahrstedt. Optimal Component Composition for Scalable Stream Processing. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ICDCS '05, 2005.

[9] G. Koch, B. Koldehofe, and K. Rothermel. Cordies: Expressive Event Correlation in Distributed Systems. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, 2010.

[10] S. Mungee, N. Surendran, and D. C. Schmidt. The Design and Performance of a CORBA Audio/Video Streaming Service. In *Hawaiian International Conference on System Sciences*, 1999.

[11] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22nd International Conference on Data Engineering*, Washington, DC, USA, 2006. IEEE Computer Society.

[12] S. Rizou, F. Dürr, and K. Rothermel. Providing QoS Guarantees in Large-Scale Operator Networks. In *12th IEEE International Conference on High Performance Computing and Communications*, 2010.

[13] S. Rizou, F. Dürr, and K. Rothermel. Solving the Multi-operator Placement Problem in Large Scale Operator Networks. In *19th Internatonal Conference on Computer Communications and Networks*, 2010.